Data Mining and Knowledge Discovery, 1, 343–373 (1997)

Parallel Algorithms for Discovery of Association Rules

MOHAMMED J. ZAKI zaki@cs.rochester.edu
SRINIVASAN PARTHASARATHY srini@cs.rochester.edu
MITSUNORI OGIHARA ogihara@cs.rochester.edu
Department of Computer Science, University of Rochester, Rochester, NY 14627

WEI LI weili@us.oracle.com
Oracle Corporation, 500 Oracle Parkway, M/S 4op9, Redwood Shores, CA 94065

Editors: Paul Stolorz and Ron Musick

Abstract. Discovery of association rules is an important data mining task. Several parallel and sequential algorithms have been proposed in the literature to solve this problem. Almost all of these algorithms make repeated passes over the database to determine the set of frequent *itemsets* (a subset of database items), thus incurring high I/O overhead. In the parallel case, most algorithms perform a sum-reduction at the end of each pass to construct the global counts, also incurring high synchronization cost.

In this paper we describe new parallel association mining algorithms. The algorithms use novel itemset clustering techniques to approximate the set of potentially maximal frequent itemsets. Once this set has been identified, the algorithms make use of efficient traversal techniques to generate the frequent itemsets contained in each cluster. We propose two clustering schemes based on equivalence classes and maximal hypergraph cliques, and study two lattice traversal techniques based on bottom-up and hybrid search. We use a vertical database layout to cluster related transactions together. The database is also selectively replicated so that the portion of the database needed for the computation of associations is local to each processor. After the initial set-up phase, the algorithms do not need any further communication or synchronization. The algorithms minimize I/O overheads by scanning the local database portion only twice. Once in the set-up phase, and once when processing the itemset clusters. Unlike previous parallel approaches, the algorithms use simple intersection operations to compute frequent itemsets and do not have to maintain or search complex hash structures.

Our experimental testbed is a 32-processor DEC Alpha cluster inter-connected by the Memory Channel network. We present results on the performance of our algorithms on various databases, and compare it against a well known parallel algorithm. The best new algorithm outperforms it by an order of magnitude.

Keywords: parallel data mining, association rules, maximal hypergraph cliques, lattice traversal

1. Introduction

With recent progress in automated data gathering, and the availability of cheap storage, a lot of businesses have routinely started collecting massive amounts of data on various facets of the organization. The eventual goal of this data gathering is to be able to use this information to gain a competitive edge, by discovering previously unknown patterns in the data, which can guide the decision making. Such high-level inference process may provide a host of useful information on customer groups, buying patterns, stock trends, etc. This process of automatic information inferencing is commonly known as Knowledge Discovery and Data mining (KDD). We look at one of the central KDD tasks — mining for

associations. Discovery of association rules is an important problem in database mining. The prototypical application is the analysis of sales or *basket* data (Agrawal, et al., 1996). Basket data consists of items bought by a customer along with the transaction identifier. Besides the retail sales example, association rules have been shown to be useful in domains such as decision support, telecommunications alarm diagnosis and prediction, university enrollments, etc.

1.1. Problem Statement

The problem of mining associations over basket data was introduced in (Agrawal, Imielinski, & Swami, 1993). It can be formally stated as: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. Each transaction T in the database $\mathcal{D}$ of transactions, has a unique identifier, and *contains* a set of items, called *itemset*, such that $T \subseteq \mathcal{I}$, i.e. each transaction is of the form $< TID, i_1, i_2, ..., i_k >$. An itemset with k items is called a k-$itemset$. A subset of length k is called a k-$subset$. An itemset is said to have a *support* s if $s\%$ of the transactions in $\mathcal{D}$ contain the itemset. An *association rule* is an expression $A \Rightarrow B$, where itemsets $A, B \subset \mathcal{I}$, and $A \cap B = \emptyset$. The *confidence* of the association rule, given as $support(A \cup B)/support(A)$, is simply the conditional probability that a transaction contains B, given that it contains A. The data mining task for association rules can be broken into two steps. The first step consists of finding all *frequent* itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called *minimum support*. The second step consists of forming conditional implication rules among the frequent itemsets (Agrawal & Srikant, 1994). The second step is relatively straightforward. Once the support of frequent itemsets is known, rules of the form $X - Y \Rightarrow Y$ (where $Y \subset X$), are generated for all frequent itemsets X, provided the rules meet a desired confidence. On the other hand the problem of identifying all frequent itemsets is hard. Given m items, there are potentially 2^m frequent itemsets, which form a *lattice of subsets* over $\mathcal{I}$. However, only a small fraction of the whole lattice space is frequent. Discovering the frequent itemsets requires a lot of computation power, memory and disk I/O, which can only be provided by parallel computers. Efficient parallel methods are needed to discover the relevant itemsets, and this is the focus of our paper.

1.2. Related Work

Sequential Algorithms Several algorithms for mining associations have been proposed in the literature. The *Apriori* algorithm (Mannila, Toivonen, & Verkamo, 1994; Agrawal & Srikant, 1994; Agrawal, et al., 1996) was shown to have superior performance to earlier approaches (Agrawal, Imielinski, & Swami, 1993; Park, Chen, & Yu, 1995a; Holsheimer, et al., 1995; Houtsma & Swami, 1995) and forms the core of almost all of the current algorithms. *Apriori* uses the *downward closure* property of itemset support to prune the itemset lattice – the property that all subsets of a frequent itemset must themselves be frequent. Thus only the frequent k-itemsets are used to construct *candidate* $(k+1)$-itemsets. A pass over the database is made at each level to find the frequent itemsets among

the candidates. For very large disk resident databases, these algorithms incur high I/O overhead for scanning it in each iteration. The *Partition* algorithm (Savasere, Omiecinski, & Navathe, 1995) minimizes I/O by scanning the database only twice. It partitions the database into small chunks which can be handled in memory. In the first pass it generates the set of all potentially frequent itemsets (any itemset locally frequent in a partition), and in the second pass their global support is obtained. Another way to minimize the I/O overhead is to work with only a small random sample of the database (Toivonen, 1996; Zaki, et al., 1997a). We recently proposed new algorithms (Zaki, et al., 1997b) which scan the database only once, generating all frequent itemsets. These new algorithms were shown to outperform previous *Apriori* based approaches by more than an order of magnitude (Zaki, et al., 1997b). The performance gains are obtained by using effective itemset clustering and lattice traversal techniques. This paper presents efficient parallel implementations of these new algorithms.

Parallel Algorithms There has been relatively less work in parallel mining of associations. Three different parallelizations of *Apriori* on a distributed-memory machine (IBM SP2) were presented in (Agrawal & Shafer, 1996). The *Count Distribution* algorithm is a straight-forward parallelization of *Apriori*. Each processor generates the partial support of all candidate itemsets from its local database partition. At the end of each iteration the global supports are generated by exchanging the partial supports among all the processors. The *Data Distribution* algorithm partitions the candidates into disjoint sets, which are assigned to different processors. However to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from huge communication overhead. The *Candidate Distribution* algorithm also partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. The local database portion is still scanned in every iteration. *Count Distribution* was shown to have superior performance among these three algorithms (Agrawal & Shafer, 1996). Other parallel algorithms improving upon these ideas in terms of communication efficiency, or aggregate memory utilization have also been proposed (Cheung, et al., 1996b; Cheung, et al., 1996a; Han, Karypis, & Kumar, 1997). The PDM algorithm (Park, Chen, & Yu, 1995b) presents a parallelization of the DHP algorithm (Park, Chen, & Yu, 1995a). However, PDM performs worse than *Count Distribution* (Agrawal & Shafer, 1996). In recent work we presented the CCPD parallel algorithm for shared-memory machines (Zaki, et al., 1996). It is similar in spirit to *Count Distribution*. The candidate itemsets are generated in parallel and are stored in a hash structure which is shared among all the processors. Each processor then scans its logical partition of the database and atomically updates the counts of candidates in the shared hash tree. CCPD uses additional optimization such as candidate balancing, hash-tree balancing and short-circuited subset counting to speed up performance (Zaki, et al., 1996). We also presented a new parallel algorithm *Eclat* (Zaki, Parthasarathy, & Li, 1997) on a DEC Alpha Cluster. *Eclat* uses the equivalence class itemset clustering scheme along with a bottom-up lattice traversal. It was shown to outperform *Count Distribution* by more than an order of magnitude. This paper will present parallelization results on new clustering and traversal techniques.

1.3. Contribution

The main limitation of all the current parallel algorithms (Park, Chen, & Yu, 1995b; Zaki, et al., 1996; Agrawal & Shafer, 1996; Cheung, et al., 1996b; Cheung, et al., 1996a) is that they make repeated passes over the disk-resident database partition, incurring high I/O overhead. Furthermore, the schemes involve exchanging either the counts of candidates or the remote database partitions during each iteration. This results in high communication and synchronization overhead. The previous algorithms also use complicated hash structures which entails additional overhead in maintaining and searching them, and typically suffer from poor cache locality (Parthasarathy, Zaki, & Li, 1997).

Our work contrasts to these approaches in several ways. We present new parallel algorithms for fast discovery of association rules based on our ideas in (Zaki, Parthasarathy, & Li, 1997; Zaki, et al., 1997b). The new parallel algorithms are characterized in terms of the clustering information used to group related itemsets, and in terms of the lattice traversal schemes used to search for frequent itemsets. We propose two clustering schemes based on equivalence classes and maximal uniform hypergraph cliques, and we utilize two lattice traversal schemes, based on bottom-up and hybrid top-down/bottom-up search. The algorithms also use a different database layout which clusters related transactions together, and the work is distributed among the processors in such a way that each processor can compute the frequent itemsets independently, using simple intersection operations. An interesting benefit of using simple intersections is that the algorithms we propose can be implemented directly on general purpose database systems (Holsheimer, et al., 1995; Houtsma & Swami, 1995). These techniques eliminate the need for synchronization after the initial set-up phase, and enable us to scan the database only two times, drastically cutting down the I/O overhead. Our experimental testbed is a 32-processor DEC Alpha SMP cluster (8 hosts, 4 processors/host) inter-connected by the Memory Channel (Gillett, 1996) network. The new parallel algorithms are also novel in that they utilize this machine configuration information, i.e., they assume a distributed-memory model across the 8 cluster hosts, but assume a shared-memory model for the 4 processors on each host. We experimentally show that our new algorithms outperform the well known *Count Distribution* algorithm. We also present extensive performance results on their speedup, sizeup, communication cost and memory usage.

The rest of the paper is organized as follows. We begin by providing more details on the sequential *Apriori* algorithm. Section 3 describes some of the previous *Apriori* based parallel algorithms. We then present the main ideas behind our new algorithms – the itemset and transaction clustering, and the lattice traversal techniques, in section 4. Section 5 describes the design and implementation of the new parallel algorithms. Our experimental study is presented in section 6, and our conclusions in section 7.

2. Sequential *Apriori* Algorithm

In this section we will briefly describe the *Apriori* algorithm (Agrawal, et al., 1996), since it forms the core of all parallel algorithms (Agrawal & Shafer, 1996; Cheung, et al., 1996b; Cheung, et al., 1996a; Han, Karypis, & Kumar, 1997; Park, Chen, & Yu, 1995b; Zaki, et

al., 1996). *Apriori* uses the downward closure property of itemset support that any subset of a frequent itemset must also be frequent. Thus during each iteration of the algorithm only the itemsets found to be frequent in the previous iteration are used to generate a new candidate set. A pruning step eliminates any candidate at least one of whose subsets is not frequent. The complete algorithm is shown in table 1. It has three main steps. The candidates for the k-th pass are generated by joining L_{k-1} with itself, which can be expressed as: $C_k = \{X = A[1]A[2]...A[k-1]B[k-1]\}$, for all $A, B \in L_{k-1}$, with $A[1:k-2] = B[1:k-2]$, and $A[k-1] < B[k-1]$, where $X[i]$ denotes the i-th item, and $X[i:j]$ denotes items at index i through j in itemset X. For example, let L_2 = {AB, AC, AD, AE, BC, BD, BE, DE}. Then C_3 = {ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE}.

Table 1. The *Apriori* Algorithm

1. $L_1 = \{$frequent 1-itemsets $\}$;
2. **for** $(k = 2; L_{k-1} \neq \emptyset; k++)$
3. $\quad C_k$ = Set of New Candidates;
4. $\quad$ **for** all transactions $t \in \mathcal{D}$
5. $\quad\quad$ **for** all k-subsets s of t
6. $\quad\quad\quad$ **if** $(s \in C_k)$ $s.count++$;
7. $\quad L_k = \{c \in C_k | c.count \geq \text{minimum support}\}$;
8. Set of all frequent itemsets = $\bigcup_k L_k$;

Before inserting an itemset into C_k, *Apriori* tests whether all its $(k-1)$-subsets are frequent. This *pruning* step can eliminate a lot of unnecessary candidates. The candidates, C_k, are stored in a hash tree to facilitate fast support counting. An internal node of the hash tree at depth d contains a hash table whose cells point to nodes at depth $d+1$. All the itemsets are stored in the leaves. The insertion procedure starts at the root, and hashing on successive items, inserts a candidate in a leaf. For counting C_k, for each transaction in the database, all k-subsets of the transaction are generated in lexicographical order. Each subset is searched in the hash tree, and the count of the candidate incremented if it matches the subset. This is the most compute intensive step of the algorithm. The last step forms L_k by selecting itemsets meeting the minimum support criterion. For details on the performance characteristics of *Apriori* we refer the reader to (Agrawal & Srikant, 1994).

3. *Apriori*-based Parallel Algorithms

In this section we will look at some previous parallel algorithms. These algorithms assume that the database is partitioned among all the processors in equal-sized blocks, which reside on the local disk of each processor.

The *Count Distribution* algorithm (Agrawal & Shafer, 1996) is a simple parallelization of *Apriori*. All processors generate the entire candidate hash tree from L_{k-1}. Each pro-

cessor can thus independently get partial supports of the candidates from its local database partition. This is followed by a sum-reduction to obtain the global counts. Note that only the partial counts need to be communicated, rather than merging different hash trees, since each processor has a copy of the entire tree. Once the global L_k has been determined each processor builds C_{k+1} in parallel, and repeats the process until all frequent itemsets are found. This simple algorithm minimizes communication since only the counts are exchanged among the processors. However, since the entire hash tree is replicated on each processor, it doesn't utilize the aggregate memory efficiently. The implementation of *Count Distribution* used for comparison in our experiments differs slightly from the above description and is optimized for our testbed configuration. Only one copy of the hash tree resides on each of the 8 hosts in our cluster. All the 4 processors on each host share this hash tree. Each processor still has its own local database portion and uses a local array to gather the local candidate support. The sum-reduction is accomplished in two steps. The first step performs the reduction only among the local processors on each host. The second step performs the reduction among the hosts. We also utilize some optimization techniques such as hash-tree balancing and short-circuited subset counting (Zaki, et al., 1996) to further improve the performance of *Count Distribution*.

The *Data Distribution* algorithm (Agrawal & Shafer, 1996) was designed to utilize the total system memory by generating disjoint candidate sets on each processor. However to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from high communication overhead, and performs very poorly when compared to *Count Distribution* (Agrawal & Shafer, 1996).

The *Candidate Distribution* algorithm (Agrawal & Shafer, 1996) uses a property of frequent itemsets (Agrawal & Shafer, 1996; Zaki, et al., 1996) to partition the candidates during iteration l, so that each processor can generate disjoint candidates independent of other processors. At the same time the database is selectively replicated so that a processor can generate global counts independently. The choice of the redistribution pass involves a trade-off between decoupling processor dependence as soon as possible and waiting until sufficient load balance can be achieved. In their experiments the repartitioning was done in the fourth pass. After this the only dependence a processor has on other processors is for pruning the candidates. Each processor asynchronously broadcasts the local frequent set to other processors during each iteration. This pruning information is used if it arrives in time, otherwise it is used in the next iteration. Note that each processor must still scan its local data once per iteration. Even though it uses problem-specific information, it performs worse than *Count Distribution* (Agrawal & Shafer, 1996). *Candidate Distribution* pays the cost of redistributing the database, and it then scans the local database partition repeatedly, which will usually be larger than $||\mathcal{D}||/P$.

4. Efficient Clustering and Traversal Techniques

In this section we present our techniques to cluster related frequent itemsets together using equivalence classes and maximal uniform hypergraph cliques. We then describe the bottom-up and hybrid itemset lattice traversal techniques. We also present a technique to cluster

related transactions together by using the vertical database layout. This layout is able to better exploit the proposed clustering and traversal schemes. It also facilitates fast itemset support counting using simple intersections, rather than maintaining and searching complex data structures.

4.1. Itemset Clustering

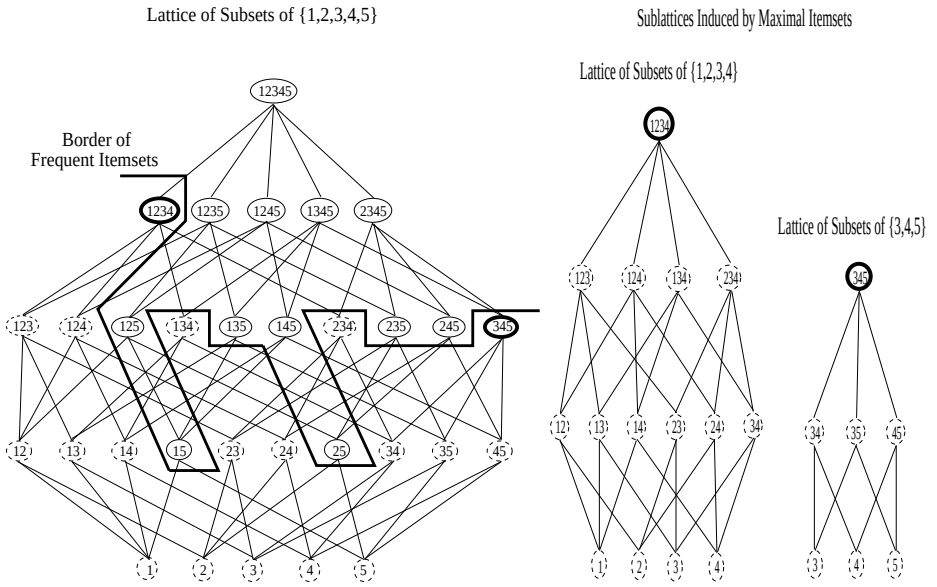

Figure 1. Lattice of Subsets and Maximal Itemset Induced Sub-lattices

We will motivate the need for itemset clustering by means of an example. Consider the lattice of subsets of the set $\{1, 2, 3, 4, 5\}$, shown in figure 1 (the empty set has been omitted in all figures). The frequent itemsets are shown with dashed circles and the two *maximal* frequent itemsets (a frequent itemset is *maximal* if it is not a proper subset of any other frequent itemset) are shown with the bold circles. Due to the downward closure property of itemset support – the fact that all subsets of a frequent itemset must be frequent – the frequent itemsets form a *border*, such that all frequent itemsets lie below the border, while all infrequent itemsets lie above it. The border of frequent itemsets is shown with a bold line in figure 1. An optimal association mining algorithm will only enumerate and test the frequent itemsets, i.e., the algorithm must efficiently determine the structure of the border. This structure is precisely determined by the maximal frequent itemsets. The border corresponds to the sub-lattices induced by the maximal frequent itemsets. These sub-lattices are shown in figure 1.

Given the knowledge of the maximal frequent itemsets we can design an efficient algorithm that simply gathers their support and the support of all their subsets in just a single database pass. In general we cannot precisely determine the maximal itemsets in the intermediate steps of the algorithm. However we can approximate this set. Our itemset clustering techniques are designed to group items together so that we obtain supersets of the maximal frequent itemsets – the *potential maximal frequent itemsets*. Below we present two schemes to generate the set of potential maximal itemsets based on equivalence classes and maximal uniform hypergraph cliques. These two techniques represent a trade-off in the precision of the potential maximal itemsets generated, and the computation cost. The hypergraph clique approach gives more precise information at higher computation cost, while the equivalence class approach sacrifices quality for a lower computation cost.

4.1.1. Equivalence Class Clustering

Let's reconsider the candidate generation step of *Apriori*. Let L_2 = {AB, AC, AD, AE, BC, BD, BE, DE}. Then C_3 = { ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE}. Assuming that L_{k-1} is lexicographically sorted, we can partition the itemsets in L_{k-1} into *equivalence classes* based on their common $k-2$ length prefixes, i.e., the equivalence class $a \in L_{k-2}$, is given as:

$$S_a = \textbf{[a]} = \{b[k-1] \in L_1 \mid \ a[1:k-2] = b[1:k-2]\}$$

Candidate k-itemsets can simply be generated from itemsets within a class by joining all $\binom{|S_i|}{2}$ pairs, with the the class identifier as the prefix. For our example L_2 above, we obtain the equivalence classes: S_A = **[A]** = {B, C, D, E}, S_B = **[B]** = {C, D, E}, and S_D = **[D]** = {E}. We observe that itemsets produced by the equivalence class **[A]**, namely those in the set {ABC, ABD, ABE, ACD, ACE, ADE}, are independent of those produced by the class **[B]** (the set {BCD, BCE, BDE}). Any class with only 1 member can be eliminated since no candidates can be generated from it. Thus we can discard the class **[D]**. This idea of partitioning L_{k-1} into equivalence classes was independently proposed in (Agrawal & Shafer, 1996; Zaki, et al., 1996). The equivalence partitioning was used in (Zaki, et al., 1996) to parallelize the candidate generation step in CCPD. It was also used in *Candidate Distribution* (Agrawal & Shafer, 1996) to partition the candidates into disjoint sets.

At any intermediate step of the algorithm when the set of frequent itemsets, L_k for $k \geq 2$, has been determined we can generate the set of potential maximal frequent itemsets from L_k. Note that for $k = 1$ we end up with the entire item universe as the maximal itemset. However, For any $k \geq 2$, we can extract more precise knowledge about the association among the items. The larger the value of k the more precise the clustering. For example, figure 2 shows the equivalence classes obtained for the instance where $k = 2$. Each equivalence class is a potential maximal frequent itemset. For example, the class **[1]**, generates the maximal itemset 12345678.

4.1.2. Maximal Uniform Hypergraph Clique Clustering

Let the set of items $\mathcal{I}$ denote the vertex set. A *hypergraph* (Berge, 1989) on $\mathcal{I}$ is a family $H = \{E_1, E_2, ..., E_n\}$ of edges or subsets of $\mathcal{I}$, such that $E_i \neq \emptyset$, and $\cup_{i=1}^{n} E_i = \mathcal{I}$. A *simple hypergraph* is a hypergraph such that, $E_i \subset E_j \Rightarrow i = j$. A simple graph is a simple hypergraph each of whose edges has cardinality 2. The maximum edge cardinality is called the *rank*, $r(H) = max_j|E_j|$. If all edges have the same cardinality, then H is called a *uniform hypergraph*. A simple uniform hypergraph of rank r is called a *r-uniform* hypergraph. For a subset $X \subset \mathcal{I}$, the *sub-hypergraph* induced by X is given as, $H_X = \{E_j \cap X \neq \emptyset | 1 \leq j \leq n\}$. A *r-uniform complete hypergraph* with m vertices, denoted as K_m^r, consists of all the r-subsets of $\mathcal{I}$. A r-uniform complete sub-hypergraph is called a *r-uniform hypergraph clique*. A hypergraph clique is *maximal* if it is not contained in any other clique. For hypergraphs of rank 2, this corresponds to the familiar concept of maximal cliques in a graph.

Given the set of frequent itemsets L_k, it is possible to further refine the clustering process producing a smaller set of potentially maximal frequent itemsets. The key observation used is that given any frequent m-itemset, for $m > k$, all its k-subsets must be frequent. In graph-theoretic terms, if each item is a vertex in the hypergraph, and each k-subset an edge, then the frequent m-itemset must form a k-uniform hypergraph clique. Furthermore, the set of maximal hypergraph cliques represents an approximation or upper-bound on the set of maximal potential frequent itemsets. All the "true" maximal frequent itemsets are contained in the vertex set of the maximal cliques, as stated formally in the lemma below.

LEMMA 1 *Let H_{L_k} be the k-uniform hypergraph with vertex set $\mathcal{I}$, and edge set L_k. Let C be the set of maximal hypergraph cliques in H, i.e., $C = \{K_m^k | m > k\}$, and let M be the set of vertex sets of the cliques in C. Then for all maximal frequent itemsets f, $\exists t \in M$, such that $f \subseteq t$.*

An example of uniform hypergraph clique clustering is given in figure 2. The example is for the case of L_2, and thus corresponds to an instance of the general clustering technique, which reduces to the case of finding maximal cliques in regular graphs. The figure shows all the equivalence classes, and the maximal cliques within them. It also shows the graph for class 1, and the maximal cliques in it. It can be seen immediately the the clique clustering is more accurate than equivalence class clustering. For example, while equivalence class clustering produced the potential maximal frequent itemset 12345678, the hypergraph clique clustering produces a more refined set $\{1235, 1258, 1278, 13456, 1568\}$ for equivalence class **[1]**. The maximal cliques are discovered using a dynamic programming algorithm. For a class **[x]**, and $y \in$**[x]**, y is said to *cover* the subset of **[x]**, given by $cov(y) = [\mathbf{y}] \cap [\mathbf{x}]$. For each class $\mathcal{C}$, we first identify its *covering set*, given as $\{y \in \mathcal{C} | cov(y) \neq \emptyset, \text{ and } cov(y) \not\subseteq cov(z), \text{ for any } z \in \mathcal{C}, z < y\}$. We recursively generate the maximal cliques for elements in the covering set for each class. Each maximal clique from the covering set is prefixed with the class identifier (eliminating any duplicates) to obtain the maximal cliques for the current class (see (Zaki, et al., 1997c) for details). For general graphs the maximal clique decision problem is NP-Complete (Garey & Johnson, 1979). However, the equivalence class graph is usually sparse and the maximal cliques can be enumerated efficiently. As

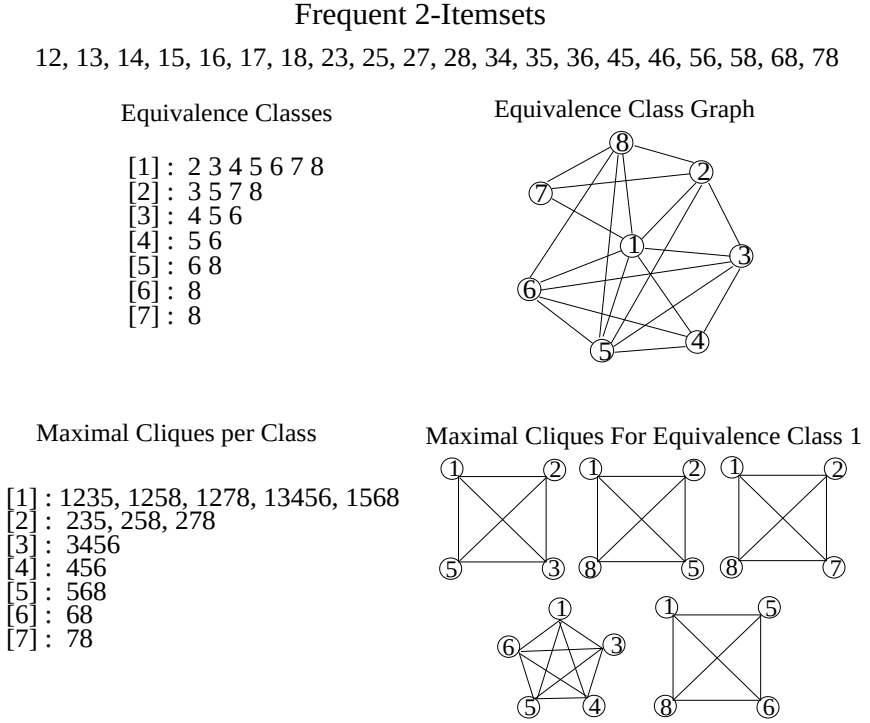

Figure 2. Equivalence Class and Uniform Hypergraph Clique Clustering

the edge density increases the clique based approaches may suffer. Some of the factors affecting the edge density include decreasing support and increasing transaction size. The effect of these parameters was presented in (Zaki, et al., 1997b).

4.2. Lattice Traversal

The equivalence class and uniform hypergraph clique clustering schemes generate the set of potential maximal frequent itemsets. Each such potential maximal itemset induces a sub-lattice of the lattice of subsets of database items $\mathcal{I}$. We now have to traverse each of these sub-lattices to determine the "true" frequent itemsets. Our goal is to devise efficient schemes to precisely determine the structure of the border of frequent itemsets. Different ways of expanding the frequent itemset border in the lattice space are possible. Below we present two schemes to traverse the sublattices. One is a purely bottom-up approach, while the other is a hybrid top-down/bottom-up scheme.

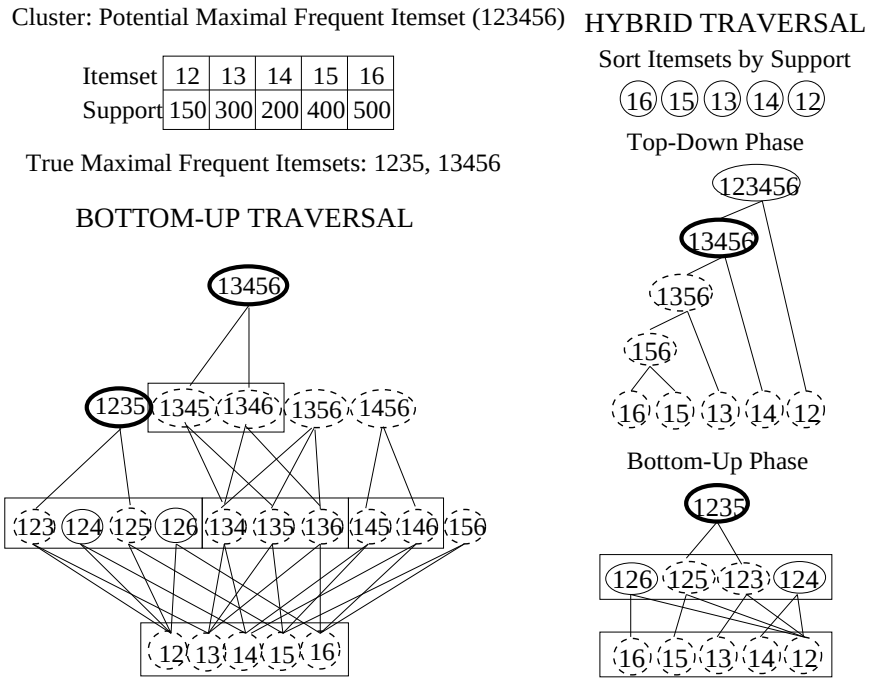

Figure 3. Bottom-up and Hybrid Lattice Traversal

4.2.1. Bottom-up Lattice Traversal

Consider the example shown in figure 3. It shows a particular instance of the clustering schemes which uses L_2 to generate the set of potential maximal itemsets. Let's assume that for equivalence class **[1]**, there is only one potential maximal itemset, 123456, while 1235 and 13456 are "true" maximal frequent itemsets. The supports of 2-itemsets in this class are also shown. Like figure 1, the dashed circles represent the frequent sets, the bold circles the maximal such itemsets, and the boxes denote equivalence classes. The potential maximal itemset 123456 forms a lattice over the elements of equivalence class **[1]** = $\{12, 13, 14, 15, 16\}$. We need to traverse this lattice to determine the "true" frequent itemsets.

A pure bottom-up lattice traversal proceeds in a breadth-first manner generating frequent itemsets of length k, before generating itemsets of level $k + 1$, i.e., at each intermediate step we determine the border of frequent k-itemsets. For example, all pairs of elements of **[1]** are joined to produce new equivalence classes of frequent 3-itemsets, namely **[12]** = $\{3, 5\}$ (producing the maximal itemset 1235), **[13]** = $\{4, 5, 6\}$, and **[14]** = $\{5, 6\}$. The next step yields the frequent class, **[134]** = $\{5, 6\}$ (producing the maximal itemset 13456). Most current algorithms use this approach. For example, the process of generating C_k from L_{k-1} used in *Apriori* (Agrawal, et al., 1996), and related algorithms (Savasere, Omiecinski,

& Navathe, 1995; Park, Chen, & Yu, 1995a), is a pure bottom-up exploration of the lattice space. Since this is a bottom-up approach all the frequent subsets of the maximal frequent itemsets are generated in intermediate steps of the traversal.

4.2.2. Hybrid Top-down/Bottom-up Search

The bottom-up approach doesn't make full use of the clustering information. While it uses the cluster to restrict the search space, it may generate spurious candidates in the intermediate steps, since the fact that all subsets of an itemset are frequent doesn't guarantee that the itemset is frequent. For example, the itemsets 124 and 126 in figure 3 are infrequent, even though 12, 14, and 16 are frequent. We can envision other traversal techniques which quickly identify the set of true maximal frequent itemsets. Once this set is known we can either choose to stop at this point if we are interested in only the maximal itemsets, or we can gather the support of all their subsets as well (all subsets are known to be frequent by definition). In this paper we will restrict our attention to only identifying the maximal frequent subsets.

One possible approach is to perform a pure top-down traversal on each cluster or sublattice. This scheme may be thought of as trying to determine the border of infrequent itemsets, by starting at the top element of the lattice and working our way down. For example, consider the potential maximal frequent itemset 123456 in figure 3. If it turns out to be frequent we are done. But in this case it is not frequent, so we then have to check whether each of its 5-subsets is frequent. At any step, if a k-subset turns out to be frequent, we need not check any of its subsets. This approach doesn't work well in practice, since the clusters are only an approximation of the maximal frequent itemsets, and a lot of infrequent supersets of the "true" maximal frequent itemsets may be generated. In our example we would generate 10 infrequent itemsets – 123456, 12345, 12346, 12356, 12456, 1234, 1245, 1236, 1246, and 1256 – using the pure top-down scheme, instead of only two infrequent itemsets generated in the pure bottom-up approach – 124, and 126. We therefore propose a hybrid top-down and bottom-up approach that works well in practice.

The basic idea behind the hybrid approach is to quickly determine the "true" maximal itemsets, by starting with a single element from a cluster of frequent k-itemsets, and extending this by one more itemset till we generate an infrequent itemset. This comprises the top-down phase. In the bottom-up phase, the remaining elements are combined with the elements in the first set to generate all the additional frequent itemsets. An important consideration in the top-down phase is to determine which elements of the cluster should be combined. In our approach we first sort the itemsets in the cluster in descending order of their support. We start with the element with maximum support, and extend it with the next element in the sorted order. This approach is based on the intuition that the larger the support the more the likely is the itemset to be a part of a larger itemset. Figure 3 shows an example of the hybrid scheme on a cluster of 2-itemsets. We sort the 2-itemsets in decreasing order of support, intersecting 16 and 15 to produce 156. This is extended to 1356 by joining 156 with 13, and then to 13456, and finally we find that 123456 is infrequent. The only remaining element is 12. We simply join this with each of the other elements producing the frequent itemset class **[12]**, which generates the other maximal itemset 1235.

The bottom-up and hybrid approaches are contrasted in figure 3, and the pseudo-code for both schemes is shown in table 3.

4.3. Transaction Clustering: Database Layout

The KDD process consists of various steps (Fayyad, Piatetsky-Shapiro, & Smyth, 1996). The initial step consists of creating the target dataset by focusing on certain attributes or via data samples. The database creation may require removing unnecessary information and supplying missing data, and transformation techniques for data reduction and projection. The user must then determine the data mining task and choose a suitable algorithm, for example, the discovery of association rules. The next step involves interpreting the discovered associations, possibly looping back to any of the previous steps, to discover more understandable patterns. An important consideration in the data preprocessing step is the final representation or data layout of the dataset. Another issue is whether some preliminary invariant information can be gleaned during this process. There are two possible layouts of the target dataset for association mining – the horizontal and the vertical layout.

4.3.1. Horizontal Data Layout

This is the format standardly used in the literature (see e.g., (Agrawal & Srikant, 1994; Mannila, Toivonen, & Verkamo, 1994; Agrawal, et al., 1996)). Here a dataset consists of a list of transactions. Each transaction has a transaction identifier (TID) followed by a list of items in that transaction. This format imposes some computation overhead during the support counting step. In particular, for each transaction of average length l, during iteration k, we have to generate and test whether all $\binom{l}{k}$ k-subsets of the transaction are contained in C_k. To perform fast subset checking the candidates are stored in a complex hash-tree data structure. Searching for the relevant candidates thus adds additional computation overhead. Furthermore, the horizontal layout forces us to scan the entire database or the local partition once in each iteration. Both *Count* and *Candidate Distribution* must pay the extra overhead entailed by using the horizontal layout. Furthermore, the horizontal layout seems suitable only for the bottom-up exploration of the frequent border. It appears to be extremely complicated to implement the hybrid approach using the horizontal format. An alternative approach is to store all the potential maximal itemsets and all their subsets in a data structure with fast look-up, (e.g., hash-trees (Agrawal, et al., 1996)). We can then gather their support in a single database scan. We plan to explore this in a later paper.

4.3.2. Vertical Data Layout

In the vertical (or inverted) layout (also called the *decomposed storage structure* (Holsheimer, et al., 1995)), a dataset consists of a list of items, with each item followed by its *tid-list* — the list of all the transactions identifiers containing the item. An example of successful use of this layout can be found in (Holsheimer, et al., 1995; Savasere, Omiecin-

Horizontal Layout

TIDS \ ITEMS	A	B	C	D	E
T1	1	0	0	1	1
T2	1	1	0	0	0
T3	0	0	1	1	1
T4	1	1	0	1	1

Vertical Layout

TIDS \ ITEMS	A	B	C	D	E
T1	1	0	0	1	1
T2	1	1	0	0	0
T3	0	0	1	1	1
T4	1	1	0	1	1

Figure 4. Horizontal and Vertical Database Layout

ski, & Navathe, 1995; Zaki, Parthasarathy, & Li, 1997; Zaki, et al., 1997b). The vertical layout doesn't suffer from any of the overheads described for the horizontal layout above due to the following three reasons: First, if the tid-list is sorted in increasing order, then the support of a candidate k-itemset can be computed by simply intersecting the tid-lists of any two $(k-1)$-subsets. No complicated data structures need to be maintained. We don't have to generate all the k-subsets of a transaction or perform the search operations on the hash tree. Second, the tid-lists contain all relevant information about an itemset, and enable us to avoid scanning the whole database to compute the support count of an itemset. This layout can therefore take advantage of the principle of locality. All frequent itemsets from a cluster of itemsets can be generated, before moving on to the next cluster. Third, the larger the itemset, the shorter the tid-lists, which is practically always true. This results in faster intersections. For example, consider figure 4, which contrasts the horizontal and the vertical layout (for simplicity, we have shown the null elements, while in reality sparse storage is used). The tid-list of A, is given as $\mathcal{T}(A) = \{1, 2, 4\}$, and $\mathcal{T}(B) = \{2, 4\}$. Then the tid-list of AB is simply, $\mathcal{T}(AB) = \{2, 4\}$. We can immediately determine the support by counting the number of elements in the tid-list. If it meets the minimum support criterion, we insert AB in L_2. The intersections among the tid-lists can be performed faster by utilizing the minimum support value. For example let's assume that the minimum support is 100, and we are intersecting two itemsets – AB with support 119 and AC with support 200. We can stop the intersection the moment we have 20 mismatches in AB, since the support of ABC is bounded above by 119. We use this optimization, called *short-circuited intersection*, for fast joins.

The inverted layout, however, has a drawback. Examination of small itemsets tends to be costlier than when the horizontal layout is employed. This is because tid-lists of small itemsets provide little information about the association among items. In particular, no such information is present in the tid-lists for 1-itemsets. For example, a database with 1,000,000 (1M) transactions, 1,000 frequent items, and an average of 10 items per transaction has tid-

lists of average size 10,000. To find frequent 2-itemsets we have to intersect each pair of items, which requires $\binom{1,000}{2} \cdot (2 \cdot 10,000) \approx 10^9$ operations. On the other hand, in the horizontal format we simply need to form all pairs of the items appearing in a transaction and increment their count, requiring only $\binom{10}{2} \cdot 1,000,000 = 4.5 \cdot 10^7$ operations.

There are a number of possible solutions to this problem:

1. To use a preprocessing step to gather the occurrence count of all 2-itemsets. Since this information is invariant, it has to be performed once during the lifetime of the database, and the cost can be amortized over the number of times the data is mined. This information can also be incrementally updated as the database changes over time.

2. To store the counts of only those 2-itemsets with support greater than a user specified lower bound, thus requiring less storage than the first approach.

3. To use a small sample that would fit in memory, and determine a superset of the frequent 2-itemsets, L_2, by lowering the minimum support, and using simple intersections on the sampled tid-lists. Sampling experiments (Toivonen, 1996; Zaki, et al., 1997a) indicate that this is a feasible approach. Once the superset has been determined we can easily verify the "true" frequent itemsets among them.

Our current implementation uses the pre-processing approach due to its simplicity. We plan to implement the sampling approach in a later paper. The solutions represent different trade-offs. The sampling approach generates L_2 on-the-fly with an extra database pass, while the pre-processing approach requires extra storage. For m items, count storage requires $\mathcal{O}(m^2)$ disk space, which can be quite large for large values of m. However, for $m = 1000$, used in our experiments this adds only a very small extra storage overhead. Using the second approach can further reduce the storage requirements, but may require an extra scan if the lower bound on support is changed. Note also that the database itself requires the same amount of memory in both the horizontal and vertical formats (this is obvious from figure 4).

5. New Parallel Algorithms: Design and Implementation

5.1. The DEC Memory Channel

Digital's Memory Channel (MC) network (Gillett, 1996) provides applications with a global address space using memory mapped regions. A region can be mapped into a process' address space for transmit, receive, or both. Virtual addresses for transmit regions map into physical addresses located in I/O space on the MC's PCI adapter. Virtual addresses for receive regions map into physical RAM. Writes into transmit regions are collected by the source MC adapter, forwarded to destination MC adapters through a hub, and transferred via DMA to receive regions with the same global identifier (see figure 5). Regions within a node can be shared across different processors on that node. Writes originating on a given node will be sent to receive regions on that same node only if *loop-back* has been enabled for the region. We do not use the loop-back feature. We use *write-doubling* instead, where

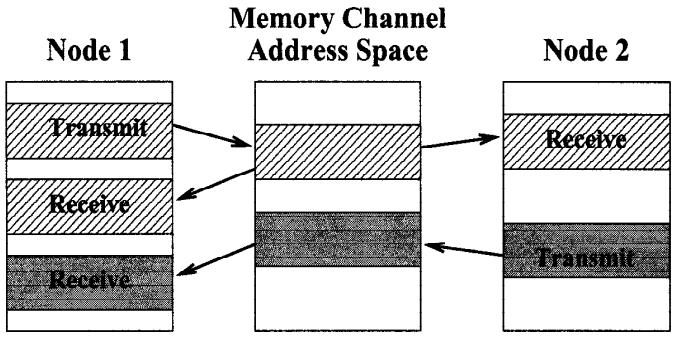

Figure 5. Memory Channel space. The lined region is mapped for both transmit and receive on node 1 and for receive on node 2. The gray region is mapped for receive on node 1 and for transmit on node 2.

each processor writes to its receive region and then to its transmit region, so that processes on a host can see modification made by other processes on the same host. Though we pay the cost of double writing, we reduce the amount of messages to the hub.

In our system unicast and multicast process-to-process writes have a latency of 5.2 μs, with per-link transfer bandwidths of 30 MB/s. MC peak aggregate bandwidth is also about 32 MB/s. Memory Channel guarantees write ordering and local cache coherence. Two writes issued to the same transmit region (even on different nodes) will appear in the same order in every receive region. When a write appears in a receive region it invalidates any locally cached copies of its line.

5.2. Initial Database Partitioning

We assume that the database is in the vertical format, and that we have the support counts of all 2-itemsets available locally on each host. We further assume that the database of tid-lists is initially partitioned among all the hosts. This partitioning is done off-line, similar to the assumption made in *Count Distribution* (Agrawal & Shafer, 1996). The tid-lists are partitioned so that the total length of all tid-lists in the local portions on each host are roughly equal. This is achieved using a greedy algorithm. The items are sorted on their support, and the next item is assigned to the least loaded host. Note that the entire tid-list for an item resides on a host. Figure 6 shows the original database, and the resultant initial partition on two processors.

5.3. New Parallel Algorithms

We present four new parallel algorithms, depending on the clustering and lattice traversal scheme used:

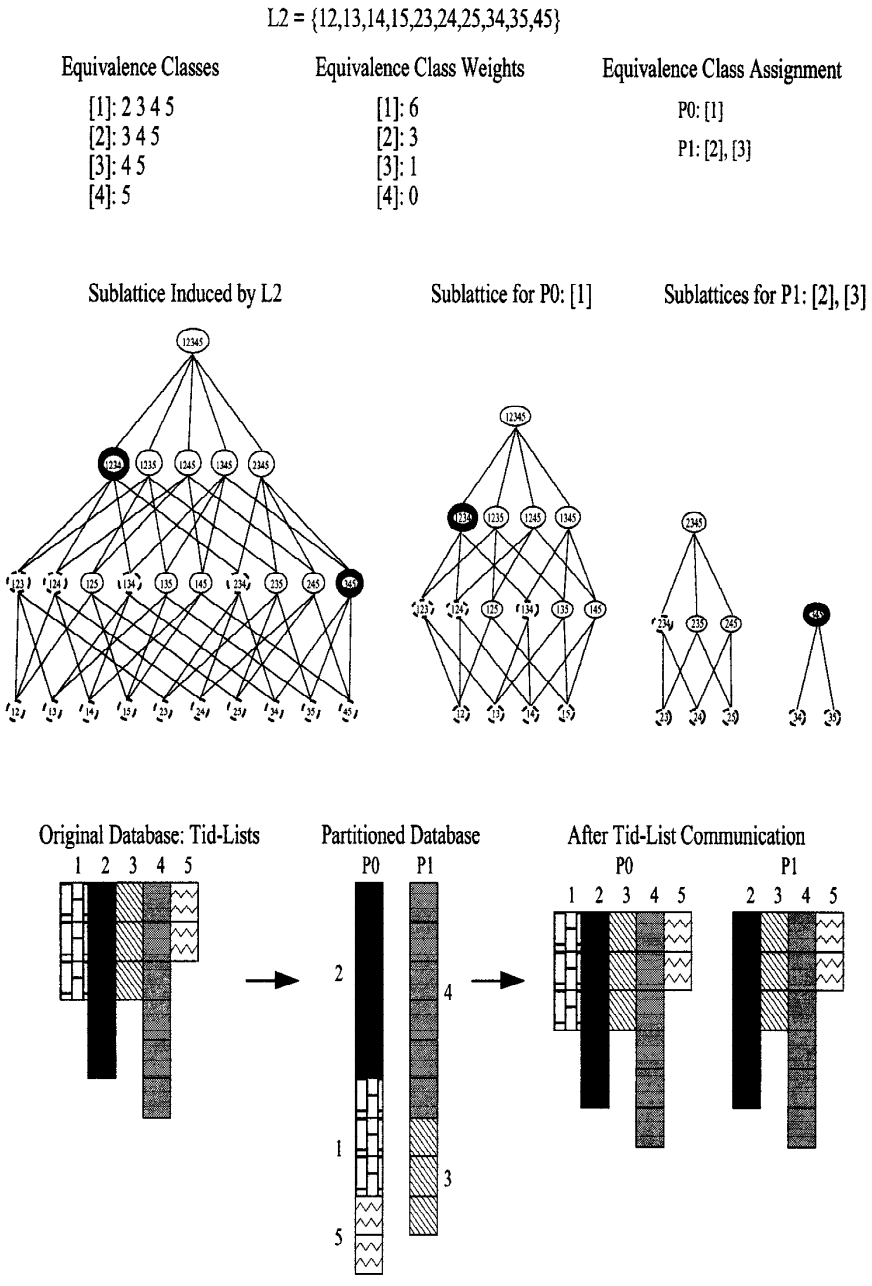

Figure 6. Database Partitioning and Cluster Scheduling

- *Par-Eclat*: uses equivalence class clustering and bottom-up lattice traversal.
- *Par-MaxEclat*: uses equivalence class clustering and hybrid traversal.
- *Par-Clique*: uses maximal uniform hypergraph clique clustering and bottom-up lattice traversal.
- *Par-MaxClique*: uses maximal uniform hypergraph clique clustering and hybrid traversal.

The algorithms using the bottom-up lattice traversal, namely *Par-Eclat* and *Par-Clique*, generate all frequent itemsets, while those using the hybrid traversal, namely *Par-MaxEclat* and *Par-MaxClique*, generate only the maximal frequent itemsets. As noted earlier, it is trivial to modify the hybrid traversal algorithms to generate all frequent itemsets. But here we are interested in examining the benefits of quickly identifying the maximal elements for the hybrid scheme. Below we present the parallel design and implementation issues, which are applicable to all four algorithms.

5.4. Parallel Design and Implementation

The new algorithms overcome the shortcomings of the *Count* and *Candidate Distribution* algorithms. They utilize the aggregate memory of the system by partitioning the itemset clusters into disjoint sets, which are assigned to different processors. The dependence among the processors is decoupled right in the beginning so that the redistribution cost can be amortized by the later iterations. Since each processor can proceed independently, there is no costly synchronization at the end of each iteration. Furthermore the new algorithms use the vertical database layout which clusters all relevant information in an itemset's tid-list. Each processor computes all the frequent itemsets from one cluster before proceeding to the next. The local database partition is scanned only once. In contrast *Candidate Distribution* must scan it once in each iteration. These algorithms don't pay the extra computation overhead of building or searching complex data structures, nor do they have to generate all the subsets of each transaction. As the intersection is performed an itemset can immediately be inserted in L_k. Notice that the tid-lists also automatically prune irrelevant transactions. As the itemset size increases, the size of the tid-list decreases, resulting in very fast intersections. There are two distinct phases in the algorithms. The initialization phase, responsible for communicating the tid-lists among the processors, and the asynchronous phase, which generates frequent itemsets. The pseudo-code for the new algorithms is shown in table 2.

5.4.1. Initialization Phase

The initialization step consists of three sub-steps. First, the support counts for 2-itemsets from the preprocessing step are read, and the frequent ones are inserted into L_2. Second, applying one of the two clustering schemes to L_2 – the equivalence class or maximal

Table 2. Pseudo-code for the New Parallel Algorithms

```
1.  Begin ParAssociation:
2.      /* Initialization Phase*/
3.      Form L_2 from 2-itemset support counts
4.      Generate Clusters from L_2 using:
5.          Equivalence Classes or Uniform Hypergraph Cliques
6.      Partition Clusters among the processors P
7.      Scan local database partition
8.      Transmit relevant tid-lists to other processors
9.      Receive tid-lists from other processors

10.     /* Asynchronous Phase */
11.     for each assigned Cluster, C_2
12.         Compute Frequent Itemsets: Bottom-Up(C_2) or Hybrid(C_2)

13.     /* Final Reduction Phase*/
14.     Aggregate Results and Output Associations
15. End ParAssociation
```

hypergraph clique clustering – the set of potential maximal frequent itemsets is generated. These potential maximal clusters are then partitioned among all the processors so that a suitable level of load-balancing can be achieved. Third, the database is repartitioned so that each processor has on its local disk the tid-lists of all 1-itemsets in any cluster assigned to it.

Cluster Scheduling We first partition the L_2 into equivalence classes using the common prefix as described above. If we are using equivalence class clustering then we already have the potential maximal itemsets. However, if we are using the clique clustering, we generate the maximal cliques within each class (see section 4). We next generate a schedule of the equivalence classes on the different processors in a manner minimizing the load imbalance and minimizing the inter-process communication. Note that it may be necessary to sacrifice some amount of load balancing for a better communication efficiency. For this reason, whole equivalence classes, including all the maximal cliques within them, are assigned to the same processor. Load balancing is achieved by assigning a weighting factor to each equivalence class based on the number of elements in the class. Since we have to consider all pairs for the next iteration, we assign the weight $\binom{s}{2}$ to a class with s elements. Once the weights are assigned we generate a schedule using a greedy heuristic. We sort the classes on the weights, and assign each class in turn to the least loaded processor, i.e., one having the least total weight at that point. Ties are broken by selecting the processor with the smaller identifier. These steps are done concurrently on all the processors since all of them have access to the global L_2. Figure 6 shows an example L_2, along with the equivalence

classes, their weights, and the assignment of the classes on two processors. Notice how an entire sublattice induced by a given class is assigned to a single processor. This leads to better load balancing, even though the partitioning may introduce extra computation. For example, if 234 were not frequent, then 1234 cannot be frequent either. But since these belong to different equivalence classes assigned to different processors, this information is not used. Although the size of a class gives a good indication of the amount of work, better heuristics for generating the weights are possible. For example, if we could better estimate the number of frequent itemsets that could be derived from an equivalence class we could use this estimation as our weight. We believe that decoupling processor performance right in the beginning holds promise, even though it may cause some load imbalance, since the repartitioning cost can be amortized over later iterations. Deriving better heuristics for scheduling the clusters, which minimize the load imbalance as well as communication, is part of ongoing research.

Tid-list Communication Once the clusters have been partitioned among the processors each processor has to exchange information with every other processor to read the non-local tid-lists over the Memory Channel network. To minimize communication, and being aware of the fact that in our configuration there is only one local disk per host (recall that our cluster has 8 hosts, with 4 processors per host), only the hosts take part in the tid-list exchange. Additional processes on each of the 8 hosts are spawned only in the asynchronous phase. To accomplish the inter-process tid-list communication, each processor scans the item tid-lists in its local database partition and writes it to a transmit region which is mapped for receive on other processors. The other processors extract the tid-list from the receive region if it belongs to any cluster assigned to them. For example, figure 6 shows the initial local database on two hosts, and the final local database after the tid-list communication.

5.4.2. Asynchronous Phase

At the end of the initialization step, the relevant tid-lists are available locally on each host, thus each processor can independently generate the frequent itemsets from its assigned maximal clusters, eliminating the need for synchronization with other processors. Each cluster is processed in its entirety before moving on to the next cluster. This step involves scanning the local database partition only once. We thus benefit from huge I/O savings. Since each cluster induces a sublattice, depending on the algorithm, we either use a bottom-up traversal to generate all frequent itemsets, or we use the hybrid traversal to generate only the maximal frequent itemsets. The pseudo-code of the two lattice traversal schemes is shown in table 3.

Note that initially we only have the tid-lists for 1-itemsets stored locally on disk. Using these, the tid-lists for the 2-itemset clusters are generated, and since these clusters are generally small the resulting tid-lists can be kept in memory. In the bottom-up approach, the tid-lists for 2-itemsets clusters are intersected to generate 3-itemsets. If the cardinality of the resulting tid-list exceeds the minimum support, the new itemset is inserted in L_3. Then we split the resulting frequent 3-itemsets, L_3 into equivalence classes based on common prefixes of length 2. All pairs of 3-itemsets within an equivalence class are intersected to determine L_4, and so on till all frequent itemsets are found. Once L_k has been determined,

Table 3. Pseudo-code for Bottom-up and Hybrid Traversal

```
1.  Input: C_k = {I_1, .., I_n}, equivalence
2.  class or maximal clique
3.  clustering of k-itemsets.
4.  Output: Frequent itemsets ∈ C_k

5.  Bottom-Up(C_k):
6.  for all I_i ∈ C_k do
7.      C_{k+1} = ∅;
8.      for all I_j ∈ C_k, i < j do
9.          N = (I_i ∩ I_j);
10.         if N.sup ≥ minsup then
11.             C_{k+1} = C_{k+1} ∪ {N};
12.     end;
13.     if C_{k+1} ≠ ∅ then
14.         Bottom-Up(C_{k+1});
15. end;
```

```
1.  Hybrid(C_2):
2.  /* Top-Down Phase */
3.  N = I_1; S_1 = {I_1};
4.  for all I_i ∈ C_2, i > 1 do
5.      N = (N ∩ I_i);
6.      if N.sup ≥ minsup then
7.          S_1 = S_1 ∪ {I_i};
8.      else break;
9.  end;
10. S_2 = C_2 − S_1;
11. /* Bottom-Up Phase */
12. for all I_i ∈ S_2,do
13.     C_3 = {(I_i ∩ X_j)|X_j ∈ S_1};
14.     S_1 = S_1 ∪ {I_i};
15.     if C_3 ≠ ∅ then Bottom-Up(C_3);
16. end;
```

we can delete L_{k-1}. We thus need main memory space only for the itemsets in L_{k-1} within one maximal cluster. For the top-down phase of the hybrid traversal only the maximal element seen so far needs to be memory-resident, along with the itemsets not yet seen. The new algorithms are therefore main memory space efficient. Experimental results on the memory usage of these algorithms are presented in the next section.

Pruning Candidates Recall that both *Count* and *Candidate Distribution* use a pruning step to eliminate unnecessary candidates. This step is essential in those algorithms to reduce the size of the hash tree. Smaller trees lead to faster support counting, since each subset of a transaction is tested against the tree. However, with the vertical database layout we found the pruning step to be of little or no help. This can be attributed to several factors. First, there is additional space and computation overhead in constructing and searching hash tables. This is also likely to degrade locality. Second, there is extra overhead in generating all the subsets of a candidate. Third, there is extra communication overhead in communicating the frequent itemsets in each iteration, even though it may happen asynchronously. Fourth, because the average size of tid-lists decreases as the itemsets size increases, intersections can be performed very quickly with the short-circuit mechanism.

At the end of the asynchronous phase we accumulate all the results from each processor and print them out.

5.5. Salient Features of the New Algorithms

In this section we will recapitulate the salient features of our proposed algorithms, contrasting them against *Count* and *Candidate Distribution*. Our algorithms differ in the following respect:

- Unlike *Count Distribution,* they utilize the aggregate memory of the parallel system by partitioning the candidate itemsets among the processors using the itemset clustering schemes.
- They decouple the processors right in the beginning by repartitioning the database, so that each processor can compute the frequent itemsets independently. This eliminates the need for communicating the frequent itemsets at the end of each iteration.
- They use the vertical database layout which clusters the transactions containing an itemset into tid-lists. Using this layout enables our algorithms to scan the local database partition only two times on each processor. The first scan for communicating the tid-lists, and the second for obtaining the frequent itemsets. In contrast, both *Count* and *Candidate Distribution* scan the database multiple times – once during each iteration.
- To compute frequent itemsets, they performs simple intersections on two tid-lists. There is no extra overhead associated with building and searching complex hash tree data structures. Such complicated hash structures also suffer from poor cache locality (Parthasarathy, Zaki, & Li, 1997). In our algorithms, all the available memory is utilized to keep tid-lists in memory which results in good locality. As larger itemsets are generated the size of tid-lists decreases, resulting in very fast intersections. Short-circuiting the join based on minimum support is also used to speed this step.
- Our algorithms avoid the overhead of generating all the subsets of a transaction and checking them against the candidate hash tree during support counting.

6. Experimental Evaluation

Table 4. Database properties

Database	T	I	$\mathcal{D}_1$	$\mathcal{D}_1$ Size	$\mathcal{D}_4$	$\mathcal{D}_4$ Size
T10.I4.D2084K	10	4	2,084,000	91 MB	8,336,000	364MB
T15.I4.D1471K	15	4	1,471,000	93 MB	5,884,000	372MB
T20.I6.D1137K	20	6	1,137,000	92 MB	4,548,000	368MB

All the experiments were performed on a 32-processor (8 hosts, 4 processors/host) Digital Alpha cluster inter-connected via the Memory Channel network (Gillett, 1996). In our system unicast and multicast process-to-process writes have a latency of 5.2 μs, with per-link transfer bandwidths of 30MB/s. Each Alpha processor runs at 233MHz. There's a

total of 256MB of main memory per host (shared among the 4 processors on that host). Each host also has a 2GB local disk attached to it, out of which less than 500MB was available to us. All the partitioned databases reside on the local disks of each processor. We used different synthetic databases, generated using the procedure described in (Agrawal & Srikant, 1994). These have been used as benchmark databases for many association rules algorithms (Agrawal & Srikant, 1994; Holsheimer, et al., 1995; Park, Chen, & Yu, 1995a; Savasere, Omiecinski, & Navathe, 1995; Agrawal, et al., 1996). Table 4 shows the databases used and their properties. The number of transactions is denoted as $\mathcal{D}_r$, where r is the replication factor. For $r = 1$, all the databases are roughly 90MB in size. Except for the sizeup experiments, all results shown are on databases with a replication factor of $r = 4$ (≈360MB). We could not go beyond a replication factor of 6 (used in sizeup experiments) since the repartitioned database would become too large to fit on disk. The average transaction size is denoted as $|T|$, and the average maximal potentially frequent itemset size as $|I|$. The number of maximal potentially frequent itemsets $|L| = 2000$, and the number of items $N = 1000$. We refer the reader to (Agrawal & Srikant, 1994) for more detail on the database generation. All the experiments were performed with a minimum support value of 0.25%. For a fair comparison, all algorithms discover frequent k-itemsets for $k \geq 3$, using the supports for the 2-itemsets from the preprocessing step.

6.1. Performance Comparison

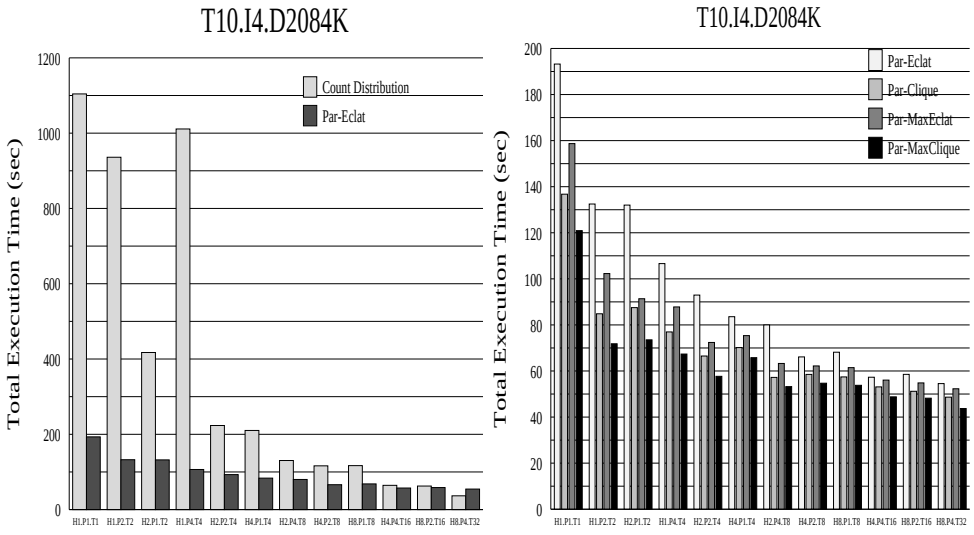

Figure 7. Parallel Performance on T10.I4.D2048K

In this section we will compare the performance of our new algorithms with *Count Distribution* (henceforth referred to as CD), which was shown to be superior to both *Data* and

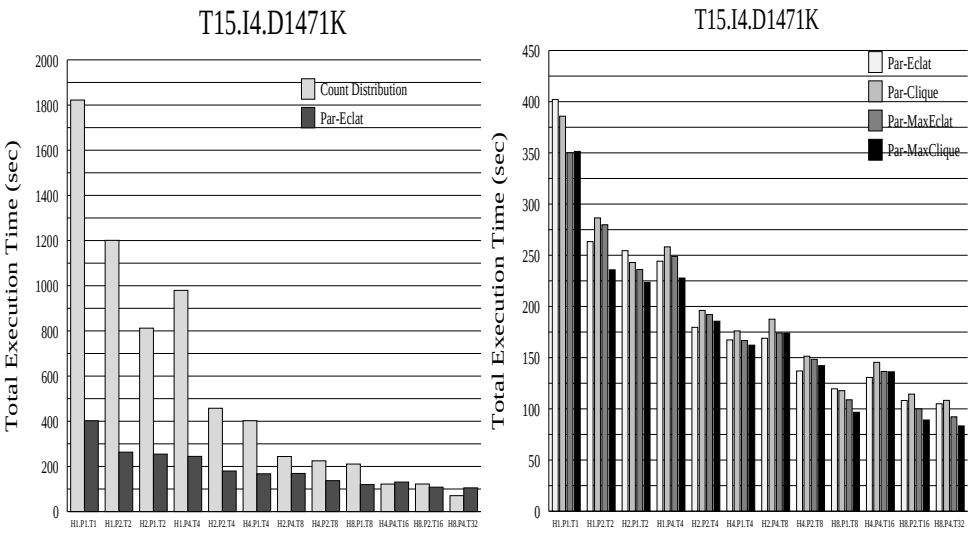

Figure 8. Parallel Performance on T15.I4.D1471K

Candidate Distribution (Agrawal & Shafer, 1996). In all the figures the different parallel configurations are represented as $Hx.Py.Tz$, where $H = x$ denotes the number of hosts, $P = y$ the number of processors per host, and $T = H \cdot P = z$, the total number of processors used in the experiments. Figures 7, 8, and 9 show the total execution time for the different databases and on different parallel configurations. The configurations have been arranged in increasing order of T. Configurations with the same T are arranged in increasing order of H. The first column compares *Par-Eclat* with CD, and the second column compares the new algorithms, so that the differences among them are more apparent. It can be clearly seen that *Par-Eclat* out-performs CD for almost all configurations on all the databases, with improvements as high as a factor of 5. If we look at the best new algorithm from the second column, we see an improvement of about an order of magnitude. Even more dramatic improvements are possible for lower minimum support (Zaki, Parthasarathy, & Li, 1997). An interesting trend in the figures is that the performance gap seems to decrease at larger configurations, with CD actually performing better at H8.P4.T32 for T10.I4.D2084K and T15.I4.D1471K. To see why consider figure 10 a, which shows the total number of frequent itemsets of different sizes for the different databases. Also from figure 11, which shows the initial database repartitioning and tid-list communication cost as a percentage of the total execution time of *Par-Eclat*, it becomes clear that there is not enough work for these two databases, to sufficiently offset the communication cost, consequently more than 70% of the time is spent in the initialization phase. For T20.I6.D1137K, which has more work, *Par-Eclat* is still about twice as fast as CD. The basic argument falls on the classic computation versus communication trade-off in parallel computing. Whenever this ratio is high we expect *Par-Eclat* to out-perform CD. We also expect the relative improvements of *Par-Eclat* over CD to be better for larger databases. Unfortunately due to disk space

constraints we were not able to test the algorithms on larger databases. In all except the $H = 1$ configurations, the local database partition is less than available memory. For CD, the entire database would be cached after the first scan. The performance of CD is thus a best case scenario for it since the results do not include the "real" hit CD would have taken from multiple disk scans. As mentioned in section 5.5, *Par-Eclat* was designed to scan the database only once during frequent itemset computation.

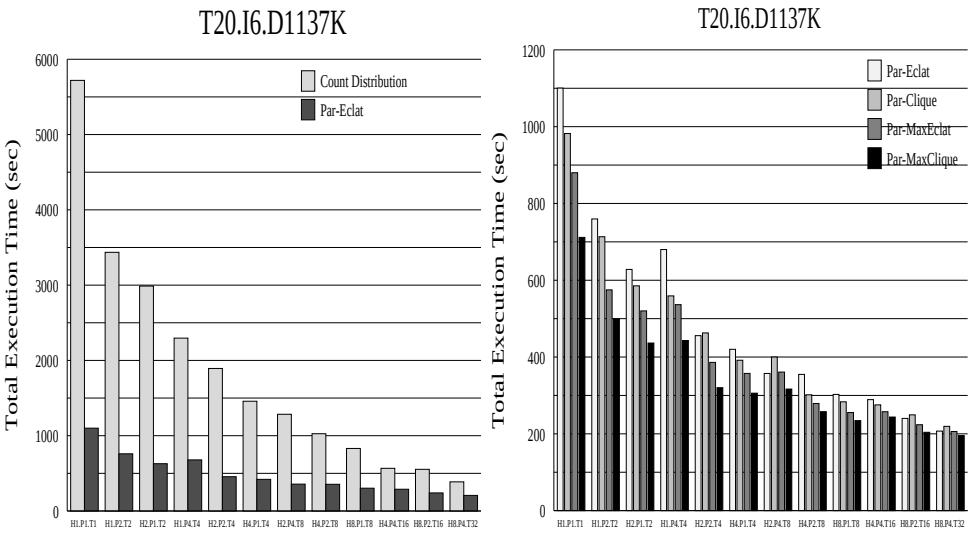

Figure 9. Parallel Performance on T20.I6.D1137K

The second column in figures 7, 8, and 9 shows the differences among the new algorithms for different databases and parallel configurations. There are several parameters affecting their performance. It can be seen that in general *Par-Clique* and *Par-MaxClique*, perform better than *Par-Eclat* and *Par-MaxEclat*, respectively. This is because they use the maximal hypergraph clique approach, which generates more precise clusters. On the other axis, in general *Par-MaxClique*, and *Par-MaxEclat*, out-perform *Par-Clique* and *Par-Eclat*, respectively. This is because the hybrid lattice traversal scheme only generates maximal frequent itemsets, saving on the number of intersections. The results are also dependent on the number of frequent itemsets. The larger the number of frequent itemsets, the more the opportunity for the hybrid approach to save on the joins. For example, consider figure 10 b, which shows the total number of tid-list intersections performed for the four algorithms on the three databases. For T20.I6.D1137K, which has the largest number of frequent itemsets (see figure 10 a), *Par-MaxClique* cuts down the number of intersections by more than 60% over *Par-Eclat*. The reduction was about 20% for *Par-MaxEclat*, and 35% for *Par-Clique*. These factors are responsible for the trends indicated above. The winner in terms of the total execution time is clearly *Par-MaxClique*, with improvements over *Par-Eclat* as high as 40%.

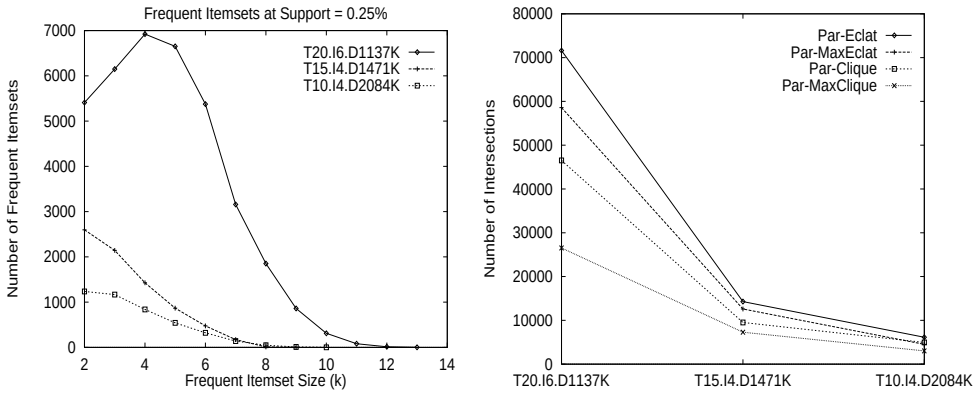

Figure 10. a) Number of Frequent k-Itemsets; b) Number of Intersections

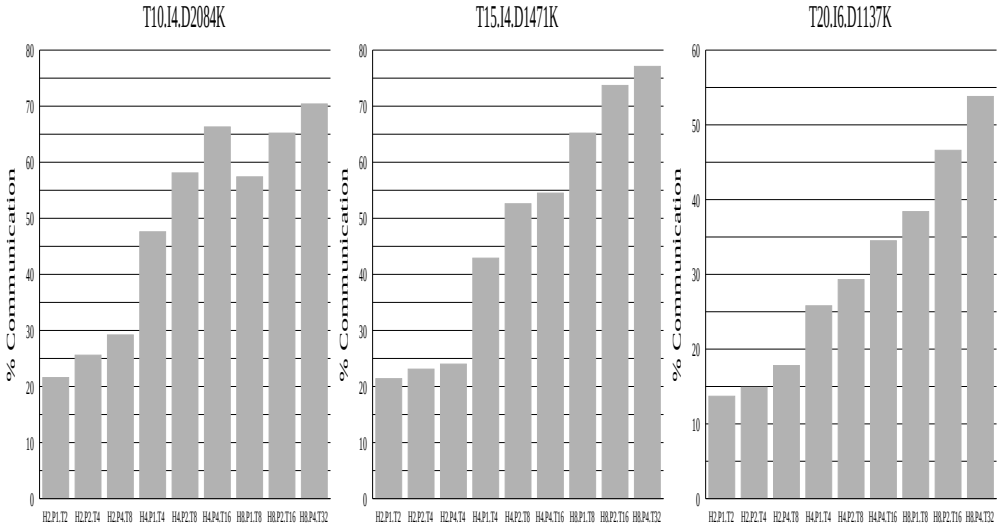

Figure 11. Communication Cost in *Par-Eclat*

6.2. Memory Usage

Figure 12 shows the total memory usage of the *Par-Eclat* algorithm as the computation of frequent itemsets progresses. The mean memory usage for the tid-lists is less than 0.7MB for all databases, even though the database itself is over 360MB. The figure only shows the cases where the memory usage was more than twice the mean. The peaks in the graph are usually due to the initial construction of all the 2-itemset tid-lists within each cluster. Since the equivalence class clusters can be large, we observe a maximum usage of 35MB for

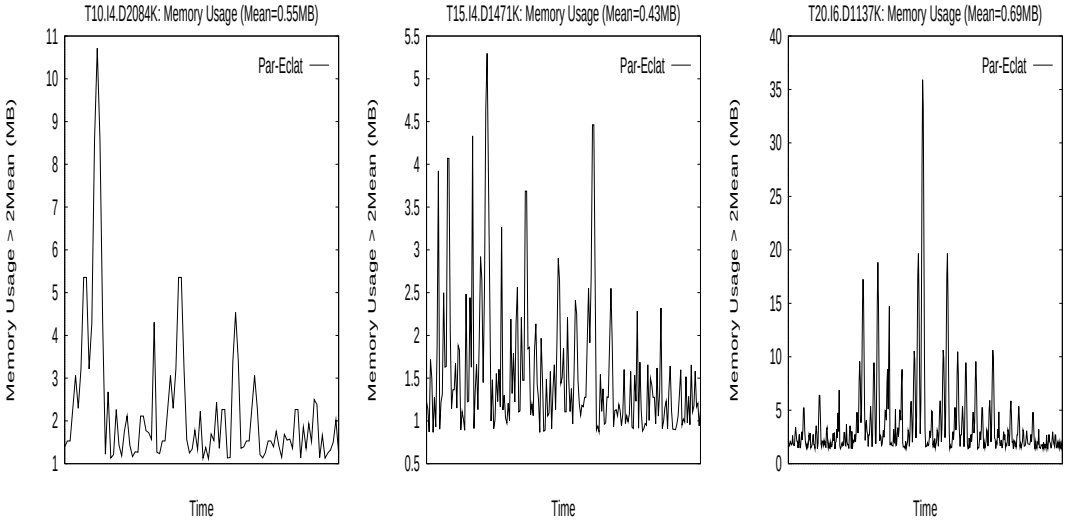

Figure 12. Memory Usage in *Par-Eclat* (H1.P1.T1)

Par-Eclat, which is still less than 10% of the database. For the other algorithms, we expect these peaks to be lower, since the maximal clique clustering is more precise, resulting in smaller clusters, and the hybrid traversal doesn't need the entire cluster 2-itemsets initially.

6.3. Sensitivity Analysis

Speedup: Figures 13, 14, and 15 (first column) show the speedup on the different databases and parallel configurations. Due to disk constraints we used a replication factor of 4, for database sizes of approximately 360MB. The speedup numbers are not as impressive at first glance. However, this is not surprising. For example, on the largest configuration H8.P4.T32, there's only about 11MB of data per processor. Combined with the fact that the amount of computation is quite small (see figure 10 a), and that about 50% to 70% of the time is spent in tid-list communication (see figure 11), we see a maximum speedup of about 5. Another reason is that the communication involves only the 8 hosts. Additional processes on a host are only spawned after the initialization phase, which thus represents a partially-parallel phase, limiting the speedups. If we take out the communication costs we see a maximum speedup of 12 to 16. An interesting trend is the step-effect seen in the speedup graphs. For the configurations which have the same number of total processors, the ones with more hosts perform better. Also, for configurations with more total processors, with $P = 4$, the configurations immediate preceding it, with only 1 processor per host, performs better. In both the cases, the reason is that increasing the number of processors on a given host, causes increased memory contention (bus traffic), and increased disk contention, as each processor tries to access the database from the local disk at the same time.

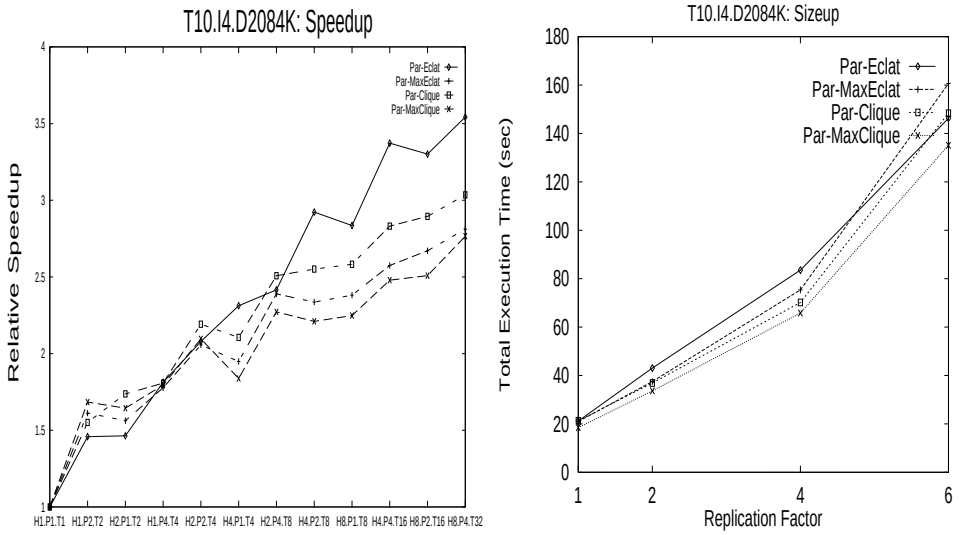

Figure 13. T10.I4.D2048K: Speedup and Sizeup(H4.P1.T4)

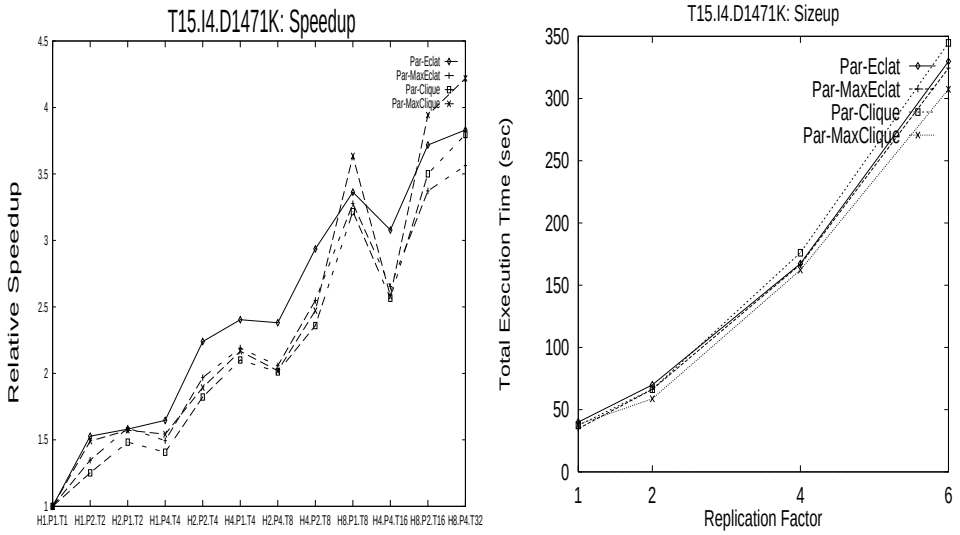

Figure 14. T15.I4.D1471K: Speedup and Sizeup(H4.P1.T4)

Sizeup: For the sizeup experiments we fixed the parallel configuration to H4.P1.T4, and varied the database replication factor from 1 to 6, with the total database size ranging from about 90MB to 540MB. Figures 13, 14, and 15 (second column) show the sizeup for the four algorithms on the different databases. The figures indicate an almost linear sizeup.

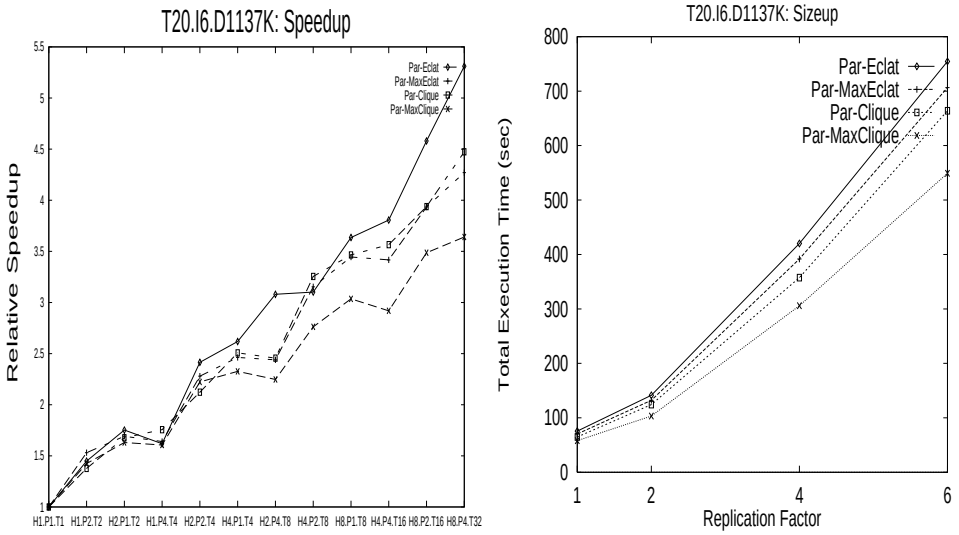

Figure 15. T20.I6.D1137K: Speedup and Sizeup(H4.P1.T4)

The slightly upward bend is due to the relative computation versus communication cost. The larger the database the more the time spent in communication, while the intersection cost doesn't increase at the same pace. Moreover, the number of frequent itemsets remains constant (since we use percentages for minimum support, as opposed to absolute counts) for all replication factors.

7. Conclusions

In this paper we proposed new parallel algorithms for the discovery of association rules. The algorithms use novel itemset clustering techniques to approximate the set of potentially maximal frequent itemsets. Once this set has been identified, the algorithms make use of efficient traversal techniques to generate the frequent itemsets contained in each cluster. We propose two clustering schemes based on equivalence classes and maximal hypergraph cliques, and study two lattice traversal techniques based on bottom-up and hybrid search. We also use the vertical database layout to cluster related transactions together. The database is also selectively replicated so that the portion of the database needed for the computation of associations is local to each processor. After the initial set-up phase, the algorithms do not need any further communication or synchronization. The algorithms minimize I/O overheads by scanning the local database portion only two times. Once in the set-up phase, and once when processing all the itemset clusters. The algorithms further use only simple intersection operations to compute frequent itemsets and don't have to maintain or search complex hash structures. An added benefit of using simple intersections is that the

algorithms we propose can be implemented directly on general purpose database systems (Holsheimer, et al., 1995; Houtsma & Swami, 1995).

Using the above techniques we presented four new algorithms. The *Par-Eclat* (equivalence class, bottom-up search) and *Par-Clique* (maximal clique, bottom-up search) algorithms, discover all frequent itemsets, while the *Par-MaxEclat* (equivalence class, hybrid search) and *Par-MaxClique* (maximal clique, hybrid search) discover the maximal frequent itemsets. We implemented the algorithms on a 32 processor DEC cluster interconnected with the DEC Memory Channel network, and compared it against a well known parallel algorithm *Count Distribution* (Agrawal & Shafer, 1996). Experimental results indicate that a substantial performance improvement is obtained using our techniques.

Acknowledgments

This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

References

Agrawal, R., and Shafer, J. 1996. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, 8(6):962–969.

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In *20th VLDB Conf.*

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast discovery of association rules. In Fayyad, U., and et al., eds., *Advances in Knowledge Discovery and Data Mining*. MIT Press.

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *ACM SIGMOD Intl. Conf. Management of Data*.

Berge, C. 1989. *Hypergraphs: Combinatorics of Finite Sets*. North-Holland.

Cheung, D.; Han, J.; Ng, V.; Fu, A.; and Fu, Y. 1996a. A fast distributed algorithm for mining association rules. *4th Intl. Conf. Parallel and Distributed Info. Systems*.

Cheung, D.; Ng, V.; Fu, A.; and Fu, Y. 1996b. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, 8(6):911–922.

Fayyad, U.; Piatetsky-Shapiro, G.; and Smyth, P. 1996. The KDD process for extracting useful knowledge from volumes of data. In *Communications of the ACM – Data Mining and Knowledge Discovery in Databases*.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co.

Gillett, R. 1996. Memory channel: An optimized cluster interconnect. In *IEEE Micro, 16(2)*.

Han, E.-H.; Karypis, G.; and Kumar, V. 1997. Scalable parallel data mining for association rules. In *ACM SIGMOD Conf. Management of Data*.

Holsheimer, M.; Kersten, M.; Mannila, H.; and Toivonen, H. 1995. A perspective on databases and data mining. In *1st Intl. Conf. Knowledge Discovery and Data Mining*.

Houtsma, M., and Swami, A. 1995. Set-oriented mining of association rules in relational databases. In *11th Intl. Conf. Data Engineering*.

Mannila, H.; Toivonen, H.; and Verkamo, I. 1994. Efficient algorithms for discovering association rules. In *AAAI Wkshp. Knowledge Discovery in Databases*.

Park, J. S.; Chen, M.; and Yu, P. S. 1995a. An effective hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*.

Park, J. S.; Chen, M.; and Yu, P. S. 1995b. Efficient parallel data mining for association rules. In *ACM Intl. Conf. Information and Knowledge Management*.

Parthasarathy, S.; Zaki, M. J.; and Li, W. 1997. Application driven memory placement for dynamic data structures. Technical Report URCS TR 653, University of Rochester.

Savasere, A.; Omiecinski, E.; and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*

Toivonen, H. 1996. Sampling large databases for association rules. In *22nd VLDB Conf.*

Zaki, M. J.; Ogihara, M.; Parthasarathy, S.; and Li, W. 1996. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing'96*.

Zaki, M. J.; Parthasarathy, S.; Li, W.; and Ogihara, M. 1997a. Evaluation of sampling for data mining of association rules. In *7th Intl. Wkshp. Research Issues in Data Engg*.

Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997b. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*.

Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997c. New algorithms for fast discovery of association rules. Technical Report URCS TR 651, University of Rochester.

Zaki, M. J.; Parthasarathy, S.; and Li, W. 1997. A localized algorithm for parallel association mining. In *9th ACM Symp. Parallel Algorithms and Architectures*.

Mohammed J. Zaki is currently completing his Ph.D. in computer science at the University of Rochester. He received a M.S. in computer science from Rochester in 1995. His research interests focus on developing efficient parallel algorithms for various data mining and knowledge discovery tasks.

Srinivasan Parthasarathy is currently a doctoral student at the University of Rochester. He received a M.S. degree in electrical engineering from the University of Cincinnati in 1994, and a M.S. degree in computer science from the University of Rochester in 1996. His research interests include parallel and distributed systems and data mining.

Mitsunori Ogihara (also known as Ogiwara) received Ph.D. in Information Sciences from Tokyo Institute of Technology in 1993. He is currently an assistant professor of computer science in the University of Rochester. His research interests are computational complexity, DNA computing, and data-mining.

Wei Li received his Ph.D. in computer science from Cornell University in 1993. He is currently with the Java Products Group in Oracle Corporation. Before that, he was an Assistant Professor at the University of Rochester. His current technical interests include Java compilation, software for network computing, and data mining.